

Floating-Point Tricks

James F. Blinn

Microsoft
Research

Oh, you work with computers? Gee, I can't even balance my checking account!"

Don't you just love it when some friend comes up with this one? So much of what we do with computers has nothing to do with arithmetic. Word processing, painting, stuff that even math-phobes can appreciate. But in the rendering game there's no substitute for nice juicy numbers. And what kind of numbers are they?

There are basically two popular numeric types inside a computer, floating point and fixed point (which I have sometimes referred to as scaled integers). In my career of number-crunching I've oscillated back and forth between favoring each of them. My first available computer (a PDP-9) didn't have floating-point hardware, so I was motivated to use scaled integer representations. Then I played with an IBM 360 and floats were practical. When I got hold of a PDP-11, floats were available but slow—back to integers. Then on to a VAX and floating point. When I moved to early PCs, it was back to scaled integers. Now that Pentia can do floating point fairly fast, it's float time again. Finally, with MMX, integers have come back to the fore. Gee, I wonder what will happen next?

Well, I recently learned about some interesting properties of the IEEE floating-point representation that sort of mixes the two. To see (and believe) how this works, we must first review this representation.

IEEE floating-point representation

The IEEE floating-point representation stores numbers in what amounts to scientific notation. Conventional scientific notation would write the number 3,485,000 in the form 3.485×10^6 . This notation is not unique; we could, for example, use any of the following:

$$3,485,500 = 3.485 \times 10^6 = .3485 \times 10^7 = \dots$$

It is typical, however, to standardize on the version with one decimal digit to the left of the decimal point.

The IEEE floating point version of scientific notation represents numbers similarly but with powers of two instead of ten. We would, for example, write

$$18_{10} = 1.001_2 \times 2^4$$

$$3.75_{10} = 1.111_2 \times 2^1$$

again standardizing on the version with one binary digit (bit) to the left of the binary point.

Next, being parsimonious about bits, the IEEE decided it didn't necessarily need to store the 1 that is just to the left of the binary point, since it's always 1. Only the fractional bits are stored in the low 23 bits of the floating-point value. The exponent of 2 is stored in the next 8 bits but offset with a bias of 127_{10} . (This allows the exponent field to be treated as an unsigned integer but still be able to represent fractions less than one). Finally, the sign bit takes up the remaining 1 bit. Some examples of this representation appear in Table 1.

Opening the box

Most programmers don't need to know any of these details. They can think of floating-point values as black boxes—subjects of mystery not to be tampered with by mere mortals. But we are not mere mortals, are we? Let's dive in.

The sign bit

In my article "Line Clipping" (*IEEE CG&A* January 1991) I opened the box for the first time. The operation in question involved creating a bit mask that specifies which clipping planes need to be processed. This decision comes from calculating a floating-point value for each vertex and for each clipping plane. If the value is

Table 1. Examples of IEEE floating-point representation.

Float	Sign	Exponent	Fraction	Combined Value Hexadecimal
.625	0	01111110	(1).010000000000000000000000	0x3F200000
1.0	0	01111111	(1).000000000000000000000000	0x3F800000
2.0	0	10000000	(1).000000000000000000000000	0x40000000
4.0	0	10000001	(1).000000000000000000000000	0x40800000
13.75	1	10000010	(1).101100000000000000000000	0xC15C0000

Table 2. Bit pattern interpretation for four-bit floating point.

Bit Pattern	Exponent		Mantissa		Floating Value Represented
	Bit Field	Interpretation	Bit Field	Interpretation	
0000	00	2^{-1}	00	$1.00_2=1.00_{10}$	$1.00 \times 2^{-1} = 0.500$
0001			01	$1.01_2=1.25_{10}$	$1.25 \times 2^{-1} = 0.625$
0010			10	$1.10_2=1.50_{10}$	$1.50 \times 2^{-1} = 0.750$
0011			11	$1.11_2=1.75_{10}$	$1.75 \times 2^{-1} = 0.875$
0100	01	2^0	00	$1.00_2=1.00_{10}$	$1.00 \times 2^0 = 1.000$
0101			01	$1.01_2=1.25_{10}$	$1.25 \times 2^0 = 1.250$
0110			10	$1.10_2=1.50_{10}$	$1.50 \times 2^0 = 1.500$
0111			11	$1.11_2=1.75_{10}$	$1.75 \times 2^0 = 1.750$
1000	10	2^1	00	$1.00_2=1.00_{10}$	$1.00 \times 2^1 = 2.000$
1001			01	$1.01_2=1.25_{10}$	$1.25 \times 2^1 = 2.500$
1010			10	$1.10_2=1.50_{10}$	$1.50 \times 2^1 = 3.000$
1011			11	$1.11_2=1.75_{10}$	$1.75 \times 2^1 = 3.500$
1100	11	2^2	00	$1.00_2=1.00_{10}$	$1.00 \times 2^2 = 4.000$
1101			01	$1.01_2=1.25_{10}$	$1.25 \times 2^2 = 5.000$
1110			10	$1.10_2=1.50_{10}$	$1.50 \times 2^2 = 6.000$
1111			11	$1.11_2=1.75_{10}$	$1.75 \times 2^2 = 7.000$

positive, the vertex is on the inside of the plane; if negative, it's on the outside of the plane. A completely legal, non-tricky way to generate the mask word would be

```
mask=0;
for (i=0,m=1; i<6; i++,m=m<<1)
    {if val[i]<0 mask|=m;}
```

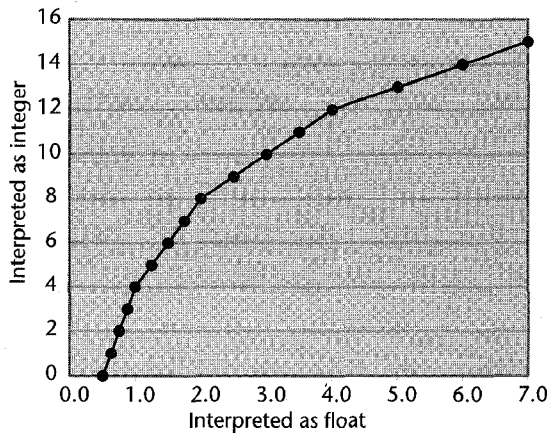
This requires six floating-point tests, which were unpleasantly slow on 1991 vintage PCs. Feeling brave, I constructed the six-bit mask by performing logical shifts on the sign bit of the floating-point words. I won't go into the details of the code, but I'll just scare you by saying that it required assembly language. The important thing, though, is that no floating-point operations were required—just logical shifting and masking of the floating-point values as bit patterns. This sort of thing might seem exceedingly machine-dependent, but let's face it, the IEEE isn't going to change their floating-point format any time soon. The only trap here is if a value happens to be minus zero (which has the perfectly reasonable representation 0x80000000), which should actually be treated as a positive number. I did get this to happen once, due to some unfortuitous forward differencing calculations. But on the whole the speed-up was worth the risk.

The rest of the bits

Now let's get to the interesting part: tricky things we can do with the rest of the floating-point word. I am indebted to Steve Gabriel and Gideon Yuval here at MS Research for first pointing out to me the following amusing fact:

If you only deal with positive numbers, the bit pattern of a floating-point number, interpreted as an integer, gives a piecewise linear approximation to the logarithm function.

To show why this is true, I'll use a rather abbreviated version of the format with 2 bits for exponent and 2 bits



1 Integer versus floating interpretation of bit pattern.

for mantissa. See Table 2.

The first thing to notice about this is that the bit pattern, interpreted as an integer, is a monotonic function of the floating-point value it represents. This means, for example, that comparing two positive floating-point numbers as floating-point numbers gives you the same answer as comparing the two bit patterns as integers. In some situations this could be handy, as, for example, when floating-point comparisons are more expensive than integer comparisons (still the case on many machines), or when you have an integer parallel processor as with MMX. Remember that this only works when the numbers are positive. An article by Walt Donovan and Tim Van Hook in *Graphics Gems IV* explores doing this with negative numbers, too ("Direct Outcode Calculation for Faster Clip Testing," *Graphics Gems IV*, P. Heckbert, ed., AP Professional, Cambridge, Mass., 1994, pp. 125-131).

The next question to ask is, what is the monotonic function? Figure 1 plots our abbreviated numbers.

Does this look familiar? It is, as I said, a piecewise linear approximation to a logarithm function. Each linear

span connects two powers of two. By properly switching interpretations of a bit pattern between integer and float, we can come up with cheap but approximate exponential and logarithmic functions.

Let's turn this into code. First, though, we need to swindle the compiler into changing the interpretation of a bit pattern between integer and float without actually doing a conversion. Two C functions that will work are

```
inline long int AsInteger(float f)
{return *(long int *)&f;}
inline float AsFloat (long int i)
{return *(float*)&i;}
```

Logarithm

Now let's do the logarithm function, in particular the base 2 logarithm. This is essentially just float (**AsInteger (x)**), but there is a scale and offset we have to include to get it right. Referring to Table 3 we can see that if we subtract the bit pattern for 1.0 (as an integer), we will get the logarithm scaled by up by 0x00800000. We simply need to divide by this quantity (as a floating value) and we're done.

Table 3. Bit patterns for base 2 logarithm.

Float	Bit Pattern	Log ₂ of Float	Bit Pattern -0x3F800000
1.0	0x3F800000	0.0	0x00000000
2.0	0x40000000	1.0	0x00800000
4.0	0x40800000	2.0	0x01000000

The code is

```
const long int OneAsInteger
= AsInteger(1.0f); // 0x3F800000
const float ScaleUp
= float(0x00800000);
const float ScaleDwn = 1./ScaleUp;

float Alog2(float x){
return float
(AsInteger(x)-OneAsInteger)
*ScaleDwn;}
```

Exponential

To get the exponential function—or, in this case, the function $y = 2^x$ —we basically reverse the above calculation.

```
float Apow2(float x){
return AsFloat
(int(x*ScaleUp)+OneAsInteger);}
```

Note that this function works properly for negative values of x .

Since the two factors **ScaleUp** and **ScaleDwn** are powers of two, it is possible to perform their multiplication by simply adding or subtracting from the exponent field of the floating-point number (again, in integer

mode). I don't do this here because a typical application will actually require powers or logarithms to some base other than 2. The conversion formulas are

$$\frac{\log_2 x}{\log_2 y} = \log_y x$$

$$2^{x \log_2 y} = y^x$$

The conversion factor $\log_2 y$, which is not a nice power of two, can be merged with the **ScaleUp** and **ScaleDwn** multiplication to generate functions for any desired base with the same number of operations as above.

Fog

One typical graphics application where the exponential function approximation is likely to be accurate enough is fog simulation. In this case the transparency of fog is given by

$$f = e^{-zd} = 2^{z(-d \log_2 e)}$$

where the fog density is d and the distance from the eye is z . If the fog density is constant, this gives another case where multiplication by the factor $-d \log_2 e$ can be merged with **ScaleUp**.

Other functions

Now we can go nuts with approximations to many other functions popular in computer graphics. One general function that we like a lot is a simple power:

$$x^p = 2^{p \log_2 x}$$

Express this with our approximation code and you get

```
float temp
= (AsInteger(x)-OneAsInteger)
*ScaleDwn;
float power
= AsFloat(int(p*temp*ScaleUp)
+OneAsInteger);}
```

The multiplications by **ScaleUp** and **ScaleDwn** cancel, and we can fold this together into

```
power = AsFloat(int(p*(AsInteger(x)
-OneAsInteger))+OneAsInteger);
```

or

```
power = AsFloat(int(p*AsInteger(x)
+(1-p)*OneAsInteger));
```

One could use this to evaluate the "cosine power" for Phong shading, but I think "cosine power" is such a bad way of modeling surface smoothness that I won't mention it further.

Square root

Here $p = 1/2$. This generates the code

```
sqrt = AsFloat((AsInteger(x)
+OneAsInteger)/2);
```

In practice, however, we must guard against integer overflow in the above addition, so I divide by two (with a shift) before adding.

Final code is

```
float Asqrt(float x){
int i = (AsInteger(x)>>1)
+(OneAsInteger>>1);
return AsFloat(i);}
```

This is actually pretty weird. We are shifting the floating-point parameter—exponent and fraction—right one bit. The low-order bit of the exponent shifts into the high-order bit of the fraction. But it works.

You can pull the expression `(OneAsInteger>>1)` out into a precalculated constant if your compiler is not smart enough to do it for you. I will not do that explicitly here or in the following code.

Inverse

This is useful in doing perspective. In this case $p = -1$. The code is

```
float Ainverse(float x){
int i = -AsInteger(x)
+ 2*OneAsInteger;
return AsFloat(i);}
```

This function also works properly for negative values of x .

Inverse square root

This is useful to normalize vectors. Here $p = -1/2$. The raw code is

```
invsqrt
= AsFloat(int(-(1/2)AsInteger(x)
+(3/2)*OneAsInteger));
```

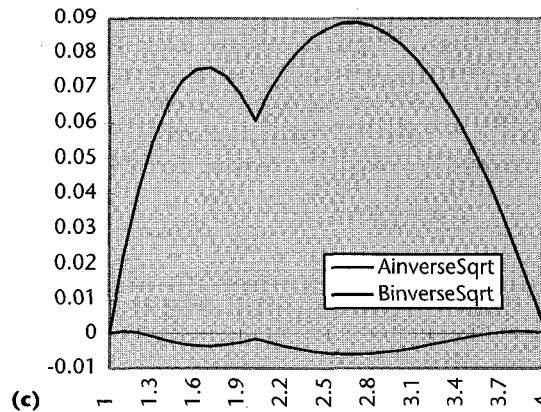
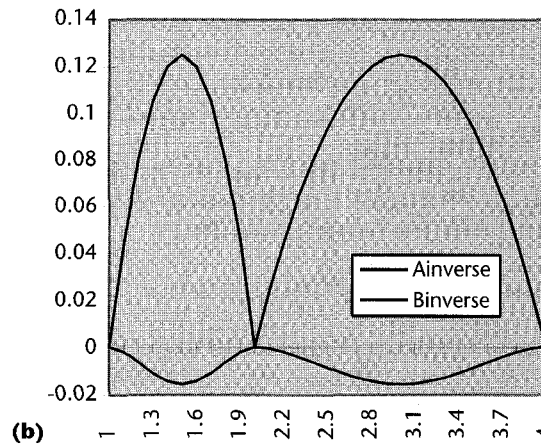
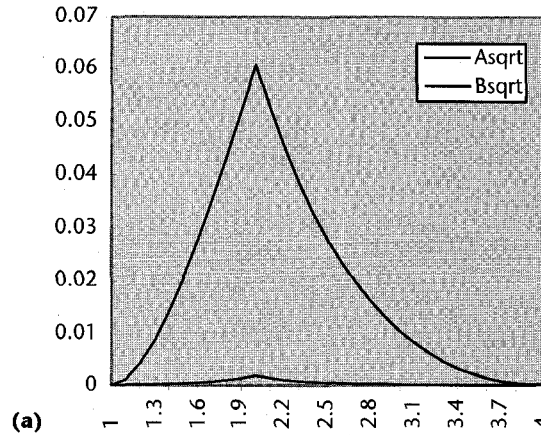
Turning this into shifts gives us the final code

```
float AinverseSqrt(float f){
int i = (OneAsInteger
+(OneAsInteger>>1))
-(AsInteger(f)>>1);
return AsFloat(i);}
```

Errors and refinements

Well, how close are these approximations? Figure 2 gives a series of graphs of the relative error (correct minus approximation divided by correct). Each error curve will repeat itself at different horizontal scales to the right and left. We can see that worst-case relative error is about 10 percent.

One nice thing about these approximations is that they make good seeds for iterative refinement techniques. The most common such technique is Newton-Raphson iteration. Simply building one step of such an iteration into the function can improve the results con-



2 Relative errors for various functions: (a) square root; (b) inverse; and (c) inverse square root.

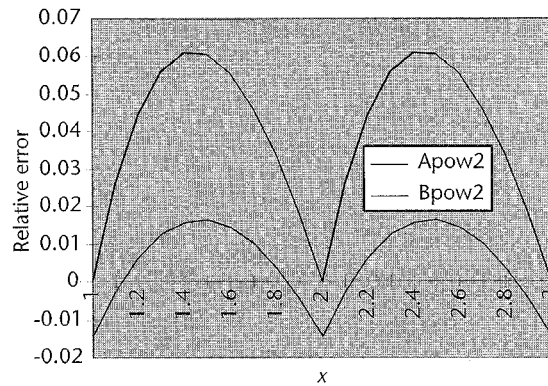
siderably. Here are better versions of our three power functions using such a scheme.

```
float Binverse(float x){
float y = Ainverse(x);
return y*(2-x*y);}
```

```
float Bsqr(float x){
float y = Asqrt(x);
return (y*y+x)/(2*y);}
```

```
float BInverseSqrt(float x){
float y = AinverseSqrt(x);
return y*(1.5-.5*x*y*y);}
```

3 Relative errors of pow2 functions.



You might be tempted to replace the division in the square root refinement with a call to `Ainverse`. When I tried this, however, the error got worse than that from the original `Asqrt` function. The relative errors of these three improved functions are plotted in the lower curves of Figure 2.

I'll tantalize you a bit by mentioning that you can double the accuracy of the inverse square root function by modifying the correction calculation to

```
return y*(1.47-.47*x*y*y);
```

That is what I actually showed in Figure 2. The explanation will have to wait for a later column, though.

How about iterative improvement of the logarithm and exponential functions? Newton iteration effectively requires evaluation of the inverse of the function we are solving for, so it's really no help here. The following trick for the exponential function comes from Gideon Yuval. First, look at the relative errors in `Apow2` plotted in the upper curve of Figure 3. The maximum errors are at points halfway between integral parameter values. Near these points we could use the mathematical identity

$$2^x = 2^{x+1/2} \times 2^{-1/2}$$

That is, we could shift our evaluation to a region where the function is more accurate by adding one half to the

parameter value and multiplying the result by a constant. But we don't really want to have to test parameter ranges and switch functions. A better plan is to calculate a sort of geometric mean between the two functions. This effectively uses the identity

$$2^x = 2^{x/2} \cdot 2^{(x/2+1/2)} \cdot 2^{-1/2}$$

Then approximate the right side with

$$2^x \approx \text{pow2}(x/2) \cdot \text{pow2}(x/2+1/2) \cdot c$$

and adjust the scale factor `c` to spread the error fairly uniformly across the `x` parameter range. Some numerical tests show that a good value is about `.657`. This gives the code

```
float Bpow2(float x) {
    return Apow2(x/2)*Apow2(x/2+.5)
        *.657;}
```

Of course, the halving of the parameter values can be absorbed into the `ScaleUp` constant within `Apow2`. The resultant relative error curve appears as the bottom curve in Figure 3.

A similar gimmick leads us to the improved version of the logarithm function

```
float Blog(float x) {
    return .5*(Alog(x)
        +Alog(x*.6666))+.344;}
```

Conclusion

For some quick and dirty approximations it pays to be brave and open the magic floating-point box. There are many other possible refinement techniques I haven't gone into here. You could, for example, try table lookups or other simple functions applied to the high few bits of the fraction. You just have to make sure your refined approximate solution doesn't wind up being slower than the original correct solution. I'm still interested in what other goodies can be extracted from this trick. ■